Use Pandas value_counts() Function (With Examples)

Authored by

Mohammed looti

November 3, 2025

The **value_counts()** function is recognized as an indispensable, fundamental utility within the powerful **pandas** library, which serves as the backbone for high-performance data manipulation and analysis in Python. Its core mission is to efficiently compute and present the frequency distribution of unique data points contained within a **pandas Series**. This function is critical for any data professional seeking to quickly understand the composition and structure of their datasets.

Determining how frequently specific observations occur is a cornerstone of **descriptive statistics** and the initial phase of **exploratory data analysis** (EDA). Whether examining categorical variables like colors or discrete numerical values like scores, `value_counts()` provides a concise, powerful, and highly performant method for data summarization. It eliminates the need for complex manual grouping or aggregation steps, delivering immediate insight into data imbalances or common occurrences. The fundamental syntax for invoking this operation is remarkably simple, designed for fluid integration into data processing pipelines:

my_series.value_counts()

In the following sections, we delve into five comprehensive examples that showcase the breadth and adaptability of the **value_counts()** method. These illustrations move beyond simple counts, covering advanced techniques such as incorporating missing values, calculating relative proportions, and intelligently grouping continuous numerical data into discrete bins.

Example 1: Calculating the Absolute Frequency of Unique Values

The most frequent and straightforward application of the `value_counts()` method involves calculating the raw, absolute count of every unique entry present within a dataset. When executed without supplying any optional arguments, the function returns a new **Series** object. This resulting Series is elegantly structured: its index consists of the unique values found in the original data, and the corresponding values represent their total frequency count.

This default behavior is essential for quickly identifying the mode (the most frequently occurring value) and understanding the overall distribution shape of the data. By default, the output is automatically sorted in descending order based on the frequency counts, ensuring that the most common items are immediately visible at the top of the resulting Series. This ordering streamlines the process of data auditing and validation.

Consider the following practical illustration, where we initialize a standard **pandas** Series containing a small collection of integers. We then apply the `value_counts()` function to generate a clear summary of the data distribution, revealing which numbers are most heavily represented:

import pandas as pd

```
#create pandas Series
my_series = pd.Series()

#count occurrences of unique values in Series
my_series.value_counts()
```

```
3 4
4 2
7 2
8 1
9 1
dtype: int64
```

The resulting output clearly articulates the distribution of values within `my_series`. The structure confirms the intrinsic sorting behavior based on the counts, moving from the highest frequency to the lowest. This simple function call efficiently transforms raw data into meaningful summary statistics:

The value 3 appears **4** times, establishing it as the most frequent entry in this dataset.
The values 4 and 7 demonstrate an equal occurrence, both appearing exactly **2** times.
The remaining data points, 8 and 9, are outliers in terms of frequency, as they each appear only once.

Example 2: Explicitly Including Missing Data (NaNs) in the Count

In the complex environment of real-world data science, encountering missing values is not just common--it is expected. These missing entries are typically represented by the standard floating-point concept of Not a Number (**NaN**). By default, the design of the **value_counts()** function dictates that it automatically ignores or excludes these missing values during the frequency calculation process.

However, analysts often require an accurate count of **NaN** entries to properly assess data quality and completeness. To override the default exclusion behavior and explicitly incorporate the count of missing values, we must leverage the optional parameter `dropna` and set its value to `False`. This crucial modification allows for a complete, holistic understanding of the data distribution, including the extent of data sparsity or incompleteness.

In the following scenario, we demonstrate this capability. We rely on the robust array manipulation capabilities of the **NumPy** library to seamlessly introduce several missing values into our existing **Series** object, thereby simulating a typical real-world dataset:

```
import pandas as pd
import numpy as np

#create pandas Series with some NaN values
my_series = pd.Series()

#count occurrences of unique values in Series, including NaNs
my_series.value_counts(dropna=False)
```

```
3.0 4
4.0 2
7.0 2
NaN 2
8.0 1
9.0 1
dtype: int64
```

By inspecting the output, we can immediately confirm the impact of setting `dropna=False`. The result now includes an entry for **NaN**, clearly indicating that there were precisely two instances of missing values embedded within the dataset. This complete frequency profile is indispensable for subsequent data imputation or cleaning efforts.

Example 3: Calculating Relative Frequency using Normalization

While absolute counts provide a measure of raw quantity, data analysts frequently require the relative frequency of values. Relative frequency expresses the count as a proportion or percentage of the total number of observations, offering context that absolute counts alone cannot provide. This proportional view is essential when comparing distributions across different datasets, especially those varying significantly in size.

Achieving this normalized distribution is simple: we pass the optional argument `normalize=True` to the `value_counts()` method. When this normalization flag is enabled, the function output transforms, returning floating-point numbers instead of integers. These float values represent the proportion of the total observations accounted for by each unique value, and mathematically, the sum of all these proportions will equate exactly to 1.0 (or 100%).

This technique is particularly valuable in statistical reporting, where percentages are often more intuitive than raw numbers for stakeholders. Below, we reuse our initial numerical Series and apply normalization to understand the proportional weight of each number:

```
import pandas as pd
```

```
#create pandas Series
my_series = pd.Series()

#count occurrences of unique values in Series
my_series.value_counts(normalize=True)
```

```
3 0.4
4 0.2
7 0.2
8 0.1
9 0.1
dtype: float64
```

The normalized output provides a clear, proportional representation of the data. Given there are 10 total observations in the **Series**, the calculation is straightforward, but the normalization handles large datasets with equal efficiency. Interpreting these results gives us the percentage breakdown:

The value 3 holds the largest weight, representing **0.4** (or 40%) of all observed values.
The value 4 represents a significant portion, equating to **0.2** (or 20%) of all values.
Similarly, the value 7 also accounts for **0.2** (or 20%) of the total observations.
The remaining values, 8 and 9, each individually account for 10% (0.1) of the observations.

Example 4: Discretization and Binning of Continuous Data

For datasets containing quantitative data--such as ages, temperatures, or financial metrics--where values are continuous rather than discrete categories, calculating the frequency of every single unique observation is often impractical and yields little meaningful insight. To address this, `value_counts()` offers the powerful `bins` argument, enabling the user to discretize the continuous range by grouping observations into a specified number of equal-sized intervals or bins, mirroring the methodology used in creating a histogram.

When an integer is passed to the `bins` argument, **pandas** intelligently calculates the appropriate bin edges. It determines the range from the minimum to the maximum value in the **Series** and divides that range into the requested number of equal-width intervals. The resulting frequency count then reflects how many observations fall into each calculated bin, transforming continuous data into useful, manageable categories.

In the subsequent example, we take our existing numerical data, which is suitable for binning despite its small size, and instruct the function to create three distinct, equal-width intervals by setting `bins=3`. This forces the method to categorize the data based on its magnitude:

```
import pandas as pd

#create pandas Series
my_series = pd.Series()

#count occurrences of unique values in Series
my_series.value_counts(bins=3)

(3.0, 5.0] 6
(5.0, 7.0] 2
(7.0, 9.0] 2
dtype: int64
```

The resulting output successfully displays the frequency count for each automatically generated interval. It is important to note the standard mathematical interval notation used: parentheses `(` indicate that the lower bound of the interval is excluded, while square brackets `]` signify that the upper bound is included in the count. This rigorous notation ensures accurate data segmentation:

The first bin, encompassing values greater than 3.0 up to and including 5.0 (i.e., 4 and 5, though only 4 is present), contains a total of **6** values.
The second bin, ranging from values greater than 5.0 up to and including 7.0 (i.e., 6 and 7, though only 7 is present), contains **2** values.
The third bin, covering values greater than 7.0 up to and including 9.0 (i.e., 8 and 9), also contains **2** values.

Example 5: Applying value_counts() to a DataFrame Column

Although `value_counts()` is fundamentally defined and implemented as a **Series** method, its most frequent application occurs within the context of a **pandas DataFrame**. Since a DataFrame is logically constructed as a collection of individual Series (where each column is a Series), applying `value_counts()` to a DataFrame requires a simple, two-step process: first, the desired column must be explicitly selected using standard indexing, which returns the underlying Series object; second, the method is then applied directly to that resultant Series.

This seamless integration allows data scientists to perform sophisticated frequency analysis on specific variables (columns) within a large, tabular dataset, making it a critical step in feature engineering and data auditing. This approach ensures that frequency analysis remains confined to the variable of interest, ignoring unrelated columns.

To illustrate, we create a small **DataFrame** modeling hypothetical sports statistics, including 'points', 'assists', and 'rebounds'. We then focus our analysis exclusively on the distribution of the

'points' column to understand player scoring frequency:

import pandas as pd

#create DataFrame
df = pd.DataFrame({'points': ,
'assists': ,
'rebounds': })
#count occurrences of unique values in 'points' column
df.value_counts()

9 3
10 2
13 1
15 1
22 1
Name: points, dtype: int64

The resulting frequency analysis output confirms that, within the 'points' column of the DataFrame, the score of 9 appears most frequently, occurring 3 times. Scores of 10 appear twice, while higher scores like 13, 15, and 22 are unique occurrences. This demonstrates the seamless capability of applying a Series-centric function to analyze variables within a larger tabular structure.

Summary and Best Practices

The **pandas** library, along with its core structural elements, the **DataFrame** and Series objects, form the foundation of most modern data science workflows. Achieving proficiency with utility functions like `value_counts()` is not merely optional--it is essential for quickly generating descriptive statistics and gaining an initial understanding of the underlying structure of any dataset.

This method provides a highly optimized and remarkably efficient pathway to summarize data distribution. Whether you are dealing with distinct categorical variables, complex continuous numerical data requiring binning, or auditing the presence of missing values (**NaN**), `value_counts()` offers a powerful, single-line solution that bypasses the need for more complex, multi-step manual aggregation or grouping operations. Consistent use of this function ensures rapid data quality assessment and insightful preliminary analysis.

For individuals seeking to further enhance their expertise in advanced data manipulation techniques using the **pandas** library, continuous consultation of official documentation and advanced tutorials is highly recommended. Mastering these foundational tools accelerates the transition from raw data collection to actionable insights.

Tutorial on using other common functions in pandas (Placeholder for actual links, keeping the original context).
Detailed documentation on frequency calculation methods.